Formal Development of an Embedded Verifier for Java Card Byte Code

Ludovic Casset
Gemplus Research Labs
ludovic.casset@gemplus.com

Lilian Burdy
Gemplus Research Labs
lilian.burdy@gemplus.com

Antoine Requet
Gemplus Research Labs
antoine.requet@gemplus.com

Abstract

The Java security policy is implemented by security components such as the Java Virtual Machine (JVM), the API, the verifier, the loader. It is of prime importance to ensure that the implementation of these components is in accordance with their specifications. Formal methods can be used to bring the mathematical proof that the implementation of these components corresponds to their specification. In this paper, a formal development is performed on the Java Card byte code verifier using the B method. The whole Java Card language is taken into account in order to provide realistic metrics on formal development. The architecture and the tricky points of the development are presented. This formalization leads to an embeddable implementation of the byte code verifier thanks to automatic code translation from formal implementation into C code. We present the formal models, discuss the integration into the card and the results of such an experiment.

1. Introduction

Smart cards have always had the reputation for being secured items of information system. These cards lock and protect their secret (data or applications). The invulnerability of smart cards comes from their conception. Everything is gathered in one single block: memory, CPU, communication, data and applications, everything holds in 25 square millimeters. Open smart cards allows to download new applications on card after its issuance. As there is no reason to believe that the downloaded code was developed following a methodology that guarantees its innocuousness, it is necessary to provide assurance to the customer that the executions of these applications are safe. That is, their execution will not threaten smart card integrity and confidentiality. The Java security policy defines the correct behavior of a program and the properties that this program must hold. For example, it is not possible to forge an integer into an object reference as Java is a type-safe language. A key point of this security policy is the byte code verifier. The aim of a byte code verifier is to statically check that the flow control and the data flow do not generate an error. Moreover, in order to perform these checks, one has to ensure the syntactical correctness of a file sent to the verifier for verification.

In this article, we describe a prototype of a formally developed and embedded byte code verifier for Java Card, a subset of the Java language adapted for smart card. This verifier relies on the Proof-Carrying Code techniques to implement a lightweight byte code type verifier. We then show that it is realistic to embed a byte code verifier into a smart card and that code developed using formal techniques and methodologies such as the B method [1] can fit the smart card constraints. This paper presents the results of one case study of the Matisse[1] project. This project aims to propose methodologies, tools and techniques for using formal methods in industrial concerns.

The remainder of this paper is organized as follow. Section 2 focuses on byte code verification principles and the Java Card context. Then section 3 emphases the model of the byte code verifier. Integration of formal development and informal development is discussed in section 4. Section 5 collects some metrics about the development and section 6 concludes.

2. Byte code verification

The byte code verification aims to enforce static constraints on downloaded byte code. Those constraints ensure that the byte code can be safely executed by the virtual machine, and cannot bypass the higher-level security mechanisms. The byte code verification is informally described in [8]. It consists in a static analysis of the downloaded applet ensuring that it conforms to

[1]European IST Project MATISSE number IST-1999-11435.

the Java Card semantics. For instance, it is checked that there are no stack overflow or underflow, that each instruction argument is of the correct type and that methods calls are performed in accordance with their visibility attributes (public, protected, etc...).

This analysis is separated in two parts: a structural verification, and a type verification. The next subsections will describe more in details the properties ensured by those verifications.

2.1. The structural verification

The structural verification consists in ensuring that the downloaded file is a valid file. That is, it really describes java classes and byte code, and the information it contains is consistent. For example, it checks that all the structures have the appropriate size and that the required parts exist. Those tests ensure that the downloaded file cannot be misinterpreted by the verifier or the virtual machine.

Apart from the purely structural tests checking the binary format, other tests more related to the content of the file are performed. Those tests ensure that there are no cycles in the inheritance hierarchy, or that no final methods are overridden.

A CAP file consists of several components that contains specific information from the Java Card package. For instance, the Method component contains the byte code of the methods, and the Class component information on classes such as references to their super classes or declared method.

Therefore in the Java Card case, we distinguish internal structural verifications from external structural verifications. The internal verifications correspond to the verifications that can be performed on a component basis. For instance, they consist in checking that the super classes occur first in the class component. The external verifications correspond to tests ensuring the consistency between components or external packages. For example, one of those tests consists in checking that the methods declared in the Class component correspond to existing methods in the Method component.

2.2. The type verification

This verification is performed on a method basis, and has to be done for each method present in the package.

The type checking part ensures that no disallowed type conversions are performed. For example, an integer cannot be converted into an object reference, downcasting can only be performed using the checkcast instruction, and arguments provided to methods have to be of compatible types.

As the type of the local variables is not explicitly stored in the byte code, it is needed to retrieve the type of those variables by analyzing the byte code. This part of the verification is the most complicated one, and is both time and memory expensive. It requires computing the type of each variable and stack element for each instruction and each execution path.

In order to make such verification possible the verification is quite conservative on the programs that are accepted. Only programs where the type of each element in the stack and local variable is the same whatever path has been taken to reach an instruction are accepted. This also requires that the size of the stack is the same for each instruction for each path that can reach this instruction.

2.3. Adaptation to embedded devices

Performing the full byte code verification requires large amount of computing power and memory. So different systems have been proposed to allow verification to be performed on highly constrained devices such as smart cards. Those systems rely on an external pre-treatment of the applet to verify. As the type verification is the most resource consuming part of the verification, they aim to simplify the verification algorithm.

Two approaches are usually used: Byte code normalisation and proof carrying code (PCC) or similar techniques. The next subsection introduces those techniques. The proof carrying code technique will be discussed more in details, since this is the approach that has been developed.

2.3.1. Byte code normalization. Byte code normalization is the approach used by Trusted-Logic's smart card verifier [7]. It consists in normalizing the verified applet so that it is simpler to verify. More exactly, the applet is modified so that each variable has one and only one type, and that the stack is empty at branch destinations. This greatly reduces the memory requirements, since the verifier does not have to keep typing information for each instruction, but only for each variable in the verified method. The computing requirements are also reduced, since only a simplified fixpoint computation has to be performed. However, as the code is modified, its size and memory requirements can theoretically increase.

2.3.2. Lightweight byte code verification. Introduced by Necula and Lee [11], proof-carrying code consists in adding a proof of the program safety to the

program. This proof can be generated by the code producer, and the code is transmitted along with its safety proof. The code receiver can then verify the proof in order to ensure the program safety. As checking the proof is simpler than generating it, the verification process can be performed by a constrained device.

An adaptation of this technique to Java has been proposed by Rose [17] and is now used by Sun's KVM [9]. In this context, the "proof" consists in additional type information corresponding to the content of local variables and stack element for the branch targets. Compared to byte code normalization, lightweight verification requires removing the jsr and ret instructions from the byte code, and needs temporary storage in EEPROM memory for storing the type information. However, lightweight verification performs the verification as a linear pass throughout the code, and leaves the code unmodified.

2.4. Formal studies on byte code verification

Most of those studies focus on the type verification part of the algorithms. One of the most complete formal models of the Java virtual machine is given by Qian [15]. He considers a large subset of the byte code and aims at proving the runtime correctness from its static typing. Then, he proposes the proof of a verifier that can be deducted from the virtual machine specification. In a more recent work [5] the authors also propose a correct implementation of almost all aspects of the Java byte code verifier. They view the verification problem as a data flow analysis, and aims to formally describe the specification to extract the corresponding code using the Specware tool.

In the Bali project, Push [14] proves a part of the JVM using the prover Isabelle/HOL [7]. Using Qian works [15], she gives the verifier specification and then proves its correctness. She also defines a subset of Java, (java [13] and aims to prove properties over it. More precisely, they formalize the type system and the semantics of this language using the Isabelle theorem prover. In a more recent work [12], Nipkow introduces the formal specification of the Java byte code verifier in Isabelle. Its idea is to come with the generic proof of the algorithm and then to instantiate it with a particular JVM.

Roses verification scheme has been proven safe using the Isabelle theorem prover by Nipkow [6], and a similar scheme for a Smart Card specific language has been proved correct using B in [16].

Works prior to the one described in this article have also been performed using the B method on the formalization of a simple verifier [4], and its implementation [3]. A similar work has been performed by Bertot [2] using the Coq theorem prover. He proves the correctness of the verification algorithm and generates an implementation using the Coq extraction mechanism. However, we seem to be the first to propose to embed a formally developed byte code verifier into a smart card.

3. Modeling a byte code verifier in B

In this section, the model of the byte code verifier is described. It is developed in two parts: The first one concerns the type verifier and the second one the structural verifier. The type verifier's development is made simpler as it relies on many services provided by the structural verifier and expressed through an interface. This latter verifier deals more with data representation and low level services provided by the card. In fact it relies on basic blocks such as the memory management, and file representation within the smart card.

3.1. The Type Verifier Model

The type verifier is entirely modeled in B, from its specification to its implementation. One proves that its implementation is consistent with its specification. The B method allows us to provide a very abstract specification that is split in several modules. In fact, we do not use a simple scheme where the specification is in the abstract machine which is refined and implemented. We provide a formal specification which is made of several modules (abstract machines, refinements and implementation). This formal specification is then refined in order to obtain an implementation.

3.1.1. The type verifier specification. The formal specification, at a very high level is very simple. It states that the verifier must return true or false. Using the refinement process, one clarifies what means returning true and what means returning false. Therefore, the specification of the type verifier is not only the abstract machine but a set of abstract machines, refinements and implementations that describes what the type verifier does. The formal specification is based on several loops for the type verification. The first loop iterates on methods contained into the CAP file being verified. Then a second loop is designed iterating on the different byte codes of the method. One only states that if a method is correct then all its byte codes are correct. Therefore, the specification remains simple. When aiming to ensure correctness of the byte code, a description of each of 184 byte codes is mandatory. This description remains abstract, specifying what each byte code does and what it modifies

(the stack, the local variable, etc ...).

3.1.2. The type verifier implementation. The formal implementation relies on properties and services defined in the formal specification. It allows to refine the final implementation. The proof process included in the development method helps ensuring that what is implemented is what has been specified. Once the proof is complete, one can have the mathematical proof that the implementation corresponds to the specification. The implementation is expressed in B0, a subset of the B language. It can then be translated into C code as explained in a next section.

When constructing the type verifier model, several services are necessary. First it is necessary to access data contained into Class, Method and Descriptor components [10]. As we are using the PCC technique, it is also necessary to access data of an additional component, i.e. the Proof component, that contains precomputed typing information. Finally, the type verifier accesses memory and has to rely on a model of memory management. These services do not need to be related directly to the type verifier. In fact, they correspond to low-level data access or to the CAP file description. An interface has been defined in order to collect the requirements and the properties on which the type verifier relies. This interface serves as a basis to construct the structural verifier described in the next subsection.

3.2. The structural verifier Model

The structural verifier implements the interface of services produced by the type verifier. It also includes internal and external tests. Internal tests correspond to tests related to each component of a CAP file, i.e. Applet, Class, etc... We refer to tests between components as external tests. External tests correspond to interdependencies between components, like shared information or references from a component to another. All the tests aim to ensure the correctness of the CAP file and consistency of data contained in the CAP file.

3.2.1. Modeling each CAP file component. We have modelled the structural verifier as a syntactical analyzer for CAP file. Therefore, each components constituting a CAP file is independently modeled. It includes all the components specified in [10]. At these standards components we add a specific custom component, i.e. the Proof Component that is relative to additional information used by the type verifier based on the PCC principles. Each component has an associated model that contains properties on their respective content and services allowing to access their content. At the very abstract level, it is not necessary to represent all the details of a component but to provide a sufficient description of its properties and its services.

Note that, even if it is really important to have a formal specification of each component, it is not necessary to have its formal implementation. In fact, as we are close to the CAP file format, it does not bring much to formally implement each component. We have decided to show that is possible to do so by formally implementing nine of the twelve components. But, errors found in formally implemented components and not formally implemented components are similar, both in terms of number and of origins. It mainly concerns errors due to the translation from informal to formal specification (wrong offset definition, lack of services). Designing the abstract machine helps to understand and clarify the informal specification. However, as one has to deal with low level implementation, it is hard to model and to implement efficiently. Moreover, the benefits are not as high as the cost for a formal development at this level.

3.2.2. Modeling test between component. The second part of the structural verifier performs the external tests. It consists in ensuring that information shared or referenced by several components are consistent. External tests are built on top of each component and relies on their correctness. Abstract machines representing external tests contain properties that must hold between components. To demonstrate the consistency between components, the model relies on two points: the properties of each component concerned by the test and their services to access to data in order to compare them. All external tests are modeled, refined and implemented in B. Formal specification allows to detect inconsistency between properties in different component and a possible lack of description.

3.2.3. Building interfaces services. The structural verifier is built in order to implements structural tests but also to provides services and properties on data contained in the CAP file. Therefore, the type verifier can rely on these properties and can use these services to access to the data. All the properties and the services required by the type verifier are collected into an dedicated interface. This latter interface is then refined and the services and properties are split among the different models of the CAP file components.

4. Integrating formal development into a smart card

This section discuss the integration of the formally developed code into a smart card operating system.

4.1. B0 to C code translator

One of the main advantages of using B method is automatic code generation. One of the main question when starting this formal development was about an efficient code generation. We have chosen to develop a simple code translator. The idea is to use it as a prototype to figure out what kind of improvements can be implemented and what kind of improvements are necessary.

The translator that we used was developed within our laboratory. It is a basic translator taking into account only implementation in B0 of the formal model. It translates B0 into C code. For an easier translation, we add types as assertion into the B0 code. It helps the translator choosing the best C type for the variable being translated. This allows, in particular, to restrict the variable memory space. For instance, an integer requires 4 bytes whereas a boolean can be represented by a single byte.

To integrate C code into a smart card chip, one can think about strong optimisation on the C code translator. In fact, most of optimisations focus on the chip itself. Therefore, there is not a great need to optimise the B0 code as it can produce standard code. Then, depending on the chip target, one can use a translator especially optimised for that chip. The advantage here is to provide a B0 code that can be translated to several chip targets without the needs to re-develop the code for each chip. Once, a translator is designed for a specific chip, it can be used intensively.

4.2. Implementing into the ATMEL AT 90 platform

The implementation we provide is on an ATMEL AT90 6464 C with 64 kB for software, 64 kB for data and 3 kB of RAM. On this platform, one aims to embed the complete verifier. With no specific optimization, the size of this prototype is to 45 kB. That includes the structural and the type verifier as well as the memory and the communication management.

The compilation chain provided by ATMEL, which has some very efficient tools such as the compiler, allows to gain a lot of space (in code size). Our goal is to demonstrate the feasibility of embedding a formal byte code verifier into a smart card. Now that we have a prototype, we can focus on optimisation.

4.3. Formal and informal development together into the smart card

The entire smart card is not modeled. Only a part of it, in our case the byte code verifier is modeled, which represents more than 90% of the code embedded into the smart card. A major question could be how far can we trust this development as not everything is formally developed. Formal development has to be used to develop complex parts of a system. It helps improving quality of products and increasing trust.

With new applications, like the verifier, comes new complexity. The help of mathematics and others tools is necessary to keep the same level of quality. The development of the verifier relies on basic blocks such as the memory management, the loading of the code. Thus, the formal code is integrated to this pre-existent and trusted part of the smart card. In order to be able to interact correctly an abstraction, a kind of interface, with these basic blocks is provided. It helps the formal development which needs information and properties about this blocks. It also helps defining accurately and with no ambiguity services required by the formal development and that have to be implemented by these blocks. Therefore, one has a gain both in terms of quality and of re-usability, as basic blocks already exist. The former benefit is obvious since re-usability is a gain of time in a development. The latter one is a consequence of the modeling. If a developer spends time in designing a formal abstraction for his basic blocks, then he can reason easily about it. Finally, errors can be discovered as well as lack of specification. All this leads to more confidence into the implementation.

5. Metrics on the byte code verifier and its development

In this section, we provide metrics about the formal development of the byte code verifier. Table 1 synthesizes metrics related to the development. In particular, one can note that the structural verifier is bigger than the type verifier. The reason is that the structural verifier contains a lot of tests, very different, that require a specification and an implementation for each one. Meanwhile, the type verifier can be seen as a single machinery including the typing rules enforced by Java Card. Moreover, the structural verifier contains services on which the type verifier relies. There is another important result. It concerns the number of generated Proof Obligations (POs). The results shows that the type verifier generate much more POs than the structural verifier. The reason is that there are much more properties in the type verifier than in the structural verifier.

Table 2 represents results on the execution of the byte code verifier on several example applets. The first three applets are example applets from Sun whereas the next two ones are more concrete ones and correspond

Table 1. Metrics on the formal development of the byte code verifier

Verifiers	Structural	Type	Total
Lines of B	35000	20000	55000
Generated POs	11700	18600	30300
Automatic proofs	81 %	72 %	75 %
Lines of C code	7540	4250	11790
Men months	8	4	12

Table 2. Metrics on the execution of the byte code verifier on some example applets

Applet	Size (kb)	Structural Verifier	Type Verifier
NullApp	4,2	50 ms	50 ms
HelloWorld	4,4	50 ms	90 ms
Wallet	3,6	100 ms	320 ms
Applet 1	5,2	150 ms	800 ms
Applet 2	8	640 ms	38 000 ms

to industrial applets already deployed in smart cards. We provide information about the size of the applet and the time needed to perform the structural and the typing verification.

The first comment one can extract from these results is that the execution of the type verifier is much more time consuming than the execution of the structural verifier. In fact, the structural verifier is a succession of tests. Some are easy to perform. Others are more tricky, but the code is only checked once. At the contrary, the type verifier needs to access data and even if the verification is linear in complexity, it requires much more time to check that the typing rules are correctly enforced.

The main result is that the time required for the verification are not so foolish. The verifier that we propose is a prototype on which we can make several optimization concerning either the efficiency and the memory foot print.

6. Conclusion

Adding an embedded byte code verifier to a Java Card allows the card to ensure its own security. It is important when one thinks about deployment architecture. Today, post-issuance on smart card, i.e. downloading new code when the card is already on the field, requires an heavy infrastructure which implies cryptographic protocols and certification center. With an on-card verifier, the deployment infrastructure is light as the card ensures its own security.

In this article, we bring the answer on two challenges: The first challenge is that, few years ago it was considered as unfeasible to embed a byte code verifier into a smart card. The second challenge concerns more methodological issues as a real on-card formally developed software has never been done before.

References

[1] J. Abrial. *The B Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Y. Bertot. A Coq Formalization of a Type Checker for Object Initialization in the Java Virtual Machine. Technical report, INRIA Sophia Antipolis, 2000.

[3] L. Casset. Formal Implementation of a Verification Algorithm Using the B Method. In *AFADL'2001*, Nancy, France, June 2001.

[4] L. Casset and J.-L. Lanet. A Formal Specication of the Java Bytecode Semantics using the B Method. In *1st ECOOP Workshop on Formal Techniques for Java Programs*, June 1999.

[5] A. Coglio, A. Goldberg, and Z. Qian. Towards a Provably-Correct Implementation of the JVM Bytecode Verifier. In *DISCEX'00*. IEEE, January 2000.

[6] G. Klein and T. Nipkow. Verified Lightweight Bytecode Verification. In *ECOOP 2000 Workshop on Formal Techniques for Java Programs*, Cannes, 2000.

[7] X. Leroy. On-Card Bytecode Verification for Java Card. In *E-smart*, Cannes, 2001.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Adison Wesley, 1996.

[9] S. Microsystems. Connected Limited Device Configuration, Specification 1.0a, J2ME, 2000.

[10] S. Microsystems. Java Card 2.1.1 Virtual Machine Specification, 2000.

[11] G. Necula and P. Lee. Proof-Carrying Code. In *POPL '97*, Paris, January 1997.

[12] T. Nipkow. Verified bytecode verifiers. In *FOSSACS 01*, LNCS 2030. Springer, 2001.

[13] T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In *Foundations of Secure Computation.* IOS Press, 2000.

[14] C. Pusch. Proving the soundness of a Java bytecode verifier in Isabelle/HOL. In *OOPSLA98 Workshop Formal Underpinnings of Java*, 1998.

[15] Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In *Formal Syntax and Semantics of Java*, LNCS 1523. Springer, 1999.

[16] A. Requet, L. Casset, and G. Grimaud. Application of the B Formal Method to the Proof of a Type Verification Algorithm. In *HASE 2000*, November 2000.

[17] E. Rose and K. H. Rose. Lightweight Bytecode Verification. In *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.